# 5 CONCLUSIONS AND FUTURE WORK

*You're bound to be unhappy if you optimize everything.*

— Donald Knuth

THE growing adoption of WebAssembly requires efficent hardening techniques. This thesis contributes to this effort with a comprehensive set of methods and tools for Software Diversification in WebAssembly. We introduce three technical contributions in this dissertation: CROW, MEWE, and WASM-MUTATE. Additionally, we present specific use cases for exploiting the diversification created for WebAssembly programs. In this chapter, we summarize the technical contributions of this dissertation, including an overview of the empirical findings of our research. Finally, we discuss future research directions in WebAssembly Software Diversification.

## 5.1 Summary of technical contributions

This thesis expands the field of Software Diversification within WebAssembly by implementing two distinct methods: compiler-based and binary-based approaches. Taking source code and LLVM bitcode as input, the compiler-based method generates WebAssembly variants. It uses enumerative synthesis and SMT solvers to produce numerous functionally equivalent variants. Importantly, these generated variants can be converted into multivariant binaries, thus enabling execution path randomization. Our compiler-based approach specializes in producing high-preservation variants. However, the use of SMT solvers for functional verification lowers the diversification speed when compared with the binary-based method. Furthermore, this method can only modify the code and function sections of WebAssembly binaries.

On the other hand, the method based on binary utilizes random e-graph traversals to create variants. This approach eliminates the need for modifications to existing compilers, ensuring compatibility with all existing WebAssembly

[0] Compilation probe time 2023/11/24 14:46:44

binaries. Additionally, it offers a swift, efficient and novel method for generating variants through inexpensive random e-graph traversals. Consequently, our binary-based approach can produce variants at a scale at least one order of magnitude larger than our compiler-based approach. The binary-based method can generate variants by transforming any segment of the Wasm binary. However, the preservation of the generated variants is lower than the compiler-based approach.

We have developed three open-source tools that are publicly accessible: CROW, MEWE, and WASM-MUTATE. CROW and MEWE utilize a compiler-based approach, whereas WASM-MUTATE employs a method based on binary. These tools automate the process of diversification, thereby increasing their practicality for deployment. At present, WASM-MUTATE is integrated in the wasmtime project[1] to improve testing. Our tools are complementary, providing combined utility. For instance, when the source code for a WebAssembly binary is unavailable, WASM-MUTATE offers an efficient solution for the generation of code variants. On the other hand, CROW and MEWE are particularly suited for scenarios that require a high level of variant preservation. Finally, one can use CROW and MEWE to generate a set of variants, which can then serve as rewriting rules for WASM-MUTATE. Moreover, when practitioners need to quickly generate variants, they could employ WASM-MUTATE, despite a potential decrease in the preservation of variants.

## 5.2 Summary of empirical findings

We demonstrate the practical application of Offensive Software Diversification in WebAssembly. In particular, we diversify 33 WebAssembly cryptomalware automatically, generating numerous variants. These variants successfully evade detection by state-of-the-art malware detection systems. Our research confirms the existence of opportunities for the malware detection community to strengthen the automatic detection of cryptojacking WebAssembly malware. Specifically, developers can improve the detection of WebAssembly malware by using multiple malware oracles. Additionally, these practitioners could employ feedback-guided diversification to identify specific transformations their implementation is susceptible to. For instance, our study found that the addition of arbitrary custom sections to WebAssembly binaries is a highly effective transformation for evading detection. This logic also applies to other transformations, such as adding unreachable code, another effective method for evading detection.

Moreover, our techniques enhance overall security from a Defensive Software Diversification perspective. We facilitate the deployment of unique, diversified and hardened WebAssembly binaries. As previously demonstrated, WebAssembly variants produced by our tools exhibit improved resistance to side-channel

[1] `https://github.com/bytecodealliance/wasm-tools`

attacks. Our tools generate variants by modifying malicious code patterns such as embedded timers used to conduct timing side-channel attacks. Simultaneously, they can produce variants that introduce noise into the execution side-channels of the original program, concurrently altering the memory layout of the JITed code generated by the host engine.

To sum up, our methods remarkably generate tens of thousands of variants in minutes. The swift production of these variants is due to the rapid transformation of WebAssembly binaries. This swift generation of variants is particularly advantageous in highly dynamic scenarios such as FaaS and CDN platforms. In this work, we do not discuss this case in depth. Yet, we have empirically tested the effectiveness of moving target defense techniques [112]. These tests were conducted on the Fastly edge computing platform. In this scenario, we incorporate multivariant executions[34]. Fastly can redeploy a WebAssembly binary across its 73 datacenters worldwide in 13 seconds on average. This enables the practical deployment of a unique variant per node using our tools. However, a 13-second window may still pose a risk despite each node potentially hosting a distinct WebAssembly variant. To mitigate this, we use multivariant binaries, invoking a unique variant with each execution. Our tools can generate dozens of unique variants every few seconds, each serving as a multivariant binary packaging thousands of other variants. This illustrates the real-world application of Defensive Software Diversification to a WebAssembly standalone scenario.

## 5.3 Future Work

Along with this dissertation we have highlighted several open challenges related to Software Diversification in WebAssembly. These challenges open up several directions for future research. In the following, we outline three concrete directions.

### 5.3.1 Improving WebAssembly malware detection via canonicalization

Malware detection is a well-known problem in the field of computer security, as outlined in works like Cohen's 1987 study on computer viruses [143]. This issue is exacerbated in environments where predictability is high and malware is expected to be replicated identically across multiple victims. In such scenarios, attackers can exploit this predictability to their advantage. For example, malicious actors could craft functionally equivalent malware to evade detection by malware detection systems. Indeed, our research has shown that employing Software Diversification can be an effective method for evading malware detection systems. This technique involves creating varied versions of a program, thereby reducing its predictability and making detection more challenging.

In our future research, we plan to tackle the challenge of refining the precision of malware detection systems. We aim to achieve this by evaluating the effectiveness of program normalization [147]. This strategy involves transforming a program into a standardized or "canonical" form [148]. A malware detection system in a pre-existing database is more likely to detect the canonical form. Just as a program can be transformed into multiple functionally equivalent variants, the inverse process is also possible. In simple terms, two functionally equivalent programs can be transformed into the same original program.

We plan to examine two strategies. First, we aim to employ WASM-MUTATE for program normalization. By reusing the e-graph data structures to use the shortest possible replacements, we can secure the canonical representation of the input program. Although normalization methods are not new, previous studies have grounded malware detection on the normalization of lifted code [149, 150]. WebAssembly does not need to be lifted, given that its binary format is innately platform-agnostic. Thus, we can directly normalize the WebAssembly binary. Secondly, we can pre-compile WebAssembly binaries at a minimal cost. Specifically, a WebAssembly binary could initially undergo JIT compilation into machine code. Overall, either of these two strategies aims to substantially reduce the number of malware variants that need consideration, thereby easing the task of classifiers in detecting harmful software.

However, these methods are not without its challenges. First, a normalization process is notably complex. The complexity of normalizing a program is comparable to that of optimizing a program. Both tasks are computing-intensive and require substantial resources. Second, canonicalization methods rely heavily on the degree to which malware variants can be normalized. If a malware variant is highly preserved, meaning it maintains its original form and structure even after compiling to machine code, it might be difficult to normalize and subsequently detect. This limitation suggests that while program normalization can significantly improve the efficiency and precision of malware detection systems, it is not a silver-bullet solution. The ability to detect highly preserved malware variants remains an area for further research.

### 5.3.2 WebAssembly dataset augmentation

In comparison to more established environments, the WebAssembly ecosystem is relatively new. According to a recent study by Hilbig et al., the global count of WebAssembly binaries is approximately 20,000 [43]. This number is minuscule when contrasted with the massive repositories of 1.5 million and 1.7 million packages in npm and PyPI, respectively. Intriguingly, this study also discloses that half of these WebAssembly binaries originated from our tools and public repositories, suggesting that the actual count of unique, real-world WebAssembly programs is just over 10,000. This scarcity of WebAssembly datasets presents considerable obstacles for machine learning analysis tools, which typically necessitate substantial data volumes for efficacious training and calibration. Software Diversification can be used as a key strategy, simulating a wide spectrum of potential software behaviors and scenarios. This objective can be achieved through strategic WebAssembly dataset augmentation.

Augmentation of program datasets has proven effective in enhancing the accuracy of classification models [151, 152, 153]. It exposes edge cases and rare conditions, thus enabling the development of better defenses against unforeseen challenges. In concrete, this approach might harden the robustness of detection systems, making them more impervious to evasion strategies. Furthermore, this strategy might facilitate the identification and reduction of biases within WebAssembly program datasets. Our future plans include leveraging our tools to bolster the training of models within the WebAssembly ecosystem, a domain where the collection of program samples is notably costly [43, 154]. Our methodologies aim to substantially increase the volume of samples available for model training, thereby enhancing the overall effectiveness and reliability of the WebAssembly ecosystem.

### 5.3.3 Code pattern feedback-guided Diversification

As presented in Chapter 4, feedback-guided diversification can facilitate the identification of specific transformations aiding particular objectives such as malware evasion and side-channel protection. On the contrary, stochastic diversification may generate variants that do not align with the specific objective. For instance, our approaches have shown less impact on ret2spec and pht side-channel attacks compared to btb attacks when using stochastic diversification. We can conceptualize this problem as an issue of diversification space exploration. By dividing the diversification search space, we can more efficiently focus the diversification process. The main benefit of this division is a narrowed search space, which can speed up and refine the process of diversification.

We intend to research into the previously mentioned concept. Our strategy could potentially incorporate feedback-directed methods that rely on specific code patterns, such as the disruption of embedded timers, to better combat side-channel attacks. For example, port contention side-channel [15] attacks can be preventing by using Software Diversification. One might count the number of instructions that result in port contention and use this data to choose or eliminate variants with a high or low count of instructions leading to port contention. Feedback-guided code pattern approaches need a comprehensive study of the mechanisms underlying various attack vectors, and the exact impact of different code transformations on them. The delicate adjustment of these methods to maintain a balance between general and specific security objectives poses a significant challenge. Continuous research and development are required to optimize and validate their effectiveness in a variety of scenarios.